How 16-bit Windows cached INI files for performance

devblogs.microsoft.com/oldnewthing/20240605-00

June 5, 2024

Raymond Chen

The INI file format served reasonably well in 16-bit Windows. Co-operative multitasking meant that the system didn't have to worry about multithreaded race conditions, and that allowed for performance optimizations.

If a program performed two INI file operations in rapid succession, the first operation would load and parse the INI file and then perform the requested read or write operation on the in-memory copy. Windows would retain the parsed (and possibly modified) INI file in memory as a one-level cache. If the program immediately performed another operation on the same INI file, Windows used the already-parsed content and thereby avoided accessing the disk.

In practice, programs tended to read and write INI file variables in bursts, so this sort of caching saved a lot of disk access. But how long was this cache valid? When was the dirty data flushed out to disk?

Operation	Cache dirty	Cache clean
Read or write from different INI file	Flush and discard	Discard
Read from same INI file	Flush and retain	Retain
Write to same INI file	Retain	Retain
Task switch	Flush and retain	Retain
Any disk I/O operation	Flush and retain	Retain

Furthermore, any disk I/O operation also marked the cache as potentially stale. Any INI file operation on a potentially stale cache first validated that the cache was still fresh by confirming that the file timestamp had not changed. If the file timestamp changed, then the code discarded the cache and loaded the file from disk. (It's not possible for a cache to be both dirty and potentially stale simultaneously.)

This system worked great in a co-operatively multitasked environment because you didn't have to worry about one thread writing to a file at the same time another thread is reading from it. It started to suffer in Windows 3.0 when it became possible to pre-emptively multitask Windows with MS-DOS applications, because that pesky race condition showed up, as well as problems like file sharing violations.

What kept the system from falling apart was that the conflict was only between Windows programs and MS-DOS programs, and MS-DOS programs did not frequently access INI files. Windows 95 managed to keep things together despite pre-emptively multitasked Win32 applications, because the INI file management code was still 16-bit and not pre-emptible.

The thing about flushing dirty caches on any disk operation is a bit of an architectural wart: It means that the file system has to call out to the INI file manager on every file operation. This is what is sometimes called a *layering violation*: A low-level component (the file system) is calling out to a high-level component (the INI file manager). Ideally, the file system shouldn't be coupled to a higher-level component like this. It means that you couldn't have an "INI file manager-less file system"; the two are now interdependent.[1]

Windows NT wanted to break free of these legacy architectural warts, and it also wanted to support multithreaded access properly, so it had to go a different way.

[1] An example of how this sort of layering violation could cause problems: The file system called out to the INI file manager to tell it to flush or invalidate caches, but that is ineffective if the INI file is on a network volume, and the file is modified by another computer on the network. That other computer isn't going to call into your computer's INI file manager to say, "Hey, I modified a file. You should invalidate your cache."